EXPERT SYSTEM VERIFICATION AND VALIDATION GUIDELINES/WORKSHOP TASK

Deliverable #1 - ES V&V Guidelines

Scott W. French
International Business Machines Corporation

September 3, 1991

Cooperative Agreement NCC 9-16
Research Activity No. AI.16

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division

ricis

Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

EXPERT SYSTEM VERIFICATION AND VALIDATION GUIDELINES/WORKSHOP TASK

Deliverable #1 - ES V&V Guidelines

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems Scott W. French of the International Business Machines Corporation. Dr. Terry Feagin and Dr. T. F. Leibfried served as RICIS research coordinators.

Funding has been provided by the Information Technology Division, Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Chris Culbert, of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

The views and conclusions contained in this report are those of the author and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.

ES V&V Guidelines/Workshop Task
RICIS Contract #69
Deliverable #1 - ES V&V Guidelines

September 3, 1991

Scott W. French

IBM Corporation
Federal Sector Division
3700 Bay Area Blvd. Houston, Texas 77508-1199

Preface

This document contains a first draft of information developed to support the ES V&V Guidelines/Workshop task under RICIS contract #69. The information contained herein is evolutionary in nature. Continuous revisions will be made based on feedback from participants in the workshop and NASA.

Knowledge Based System V&V Guidelines Workshop

Contents

Introduction

- Goals

  1. To show that verifying and validating a software system is a required part of developing software
  2. To show that verifying and validating a software system directly impacts its design and structure

- Justification of Goal 1

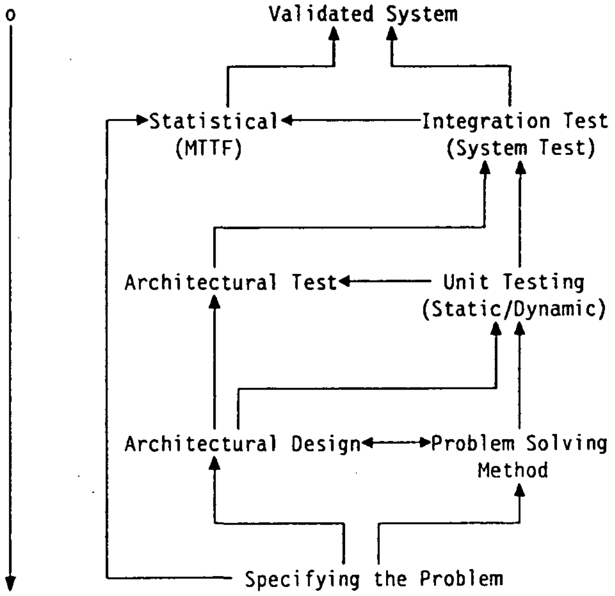
  - ► Validating software is in the interest of the user/customer
    - – Incorporates the user/customer perspective
      - ↦ Does this system solve the problem - un-validated systems provide fuzzy answers to this question
    - – Installation and checkout can be done reliably
    - – System reacts in a predictable fashion
      - ↦ Less risk that a failure in the system could pollute other un-related processes
      - ↦ Other processes can interact with the embedded system in predictable ways.

Introduction

- <u>Justification of Goal 1</u>
  - ► Verifying software is in the interest of the developer
    - – Saves money over the life-cycle of the product
      - ↦ development is easier
      - ↦ maintenance is easier
      - ↦ easier to manage

Introduction

- Justification of Goal 2
  - ► Technique:
    - — Describe how a software system can be validated
    - — Separate this description from the similar description of designing a software system
  - ► Perspective:
    - — Approach this purpose based on the goal
      - ↦ Given that validated software is the goal ⟶ What should be done to meet the goal?



    - — Avoid advocating a particular process model

Introduction

- Justification of Goal 2
  - ► Discussion
    - – Identify 7 major Tasks to satisfy the goal of validating a software system
      - ↦ What are the characteristics?
      - ↦ What are the inputs?
      - ↦ What are the implications?
    - – Identify Techniques to support implementing those Tasks
      - ↦ Special emphasis on finding errors early in the process
        - Cheaper to fix
        - Eases verification burden of later tasks

Key Terms

Dynamic Testing A type of testing involving execution of the software and analysis of the results of that execution

Module A conceptual element that captures a set of states along with specific operations for changing state and accessing state information

State Data that persists over time

Static Testing A type of testing done via inspection of the software as opposed to execution of the software

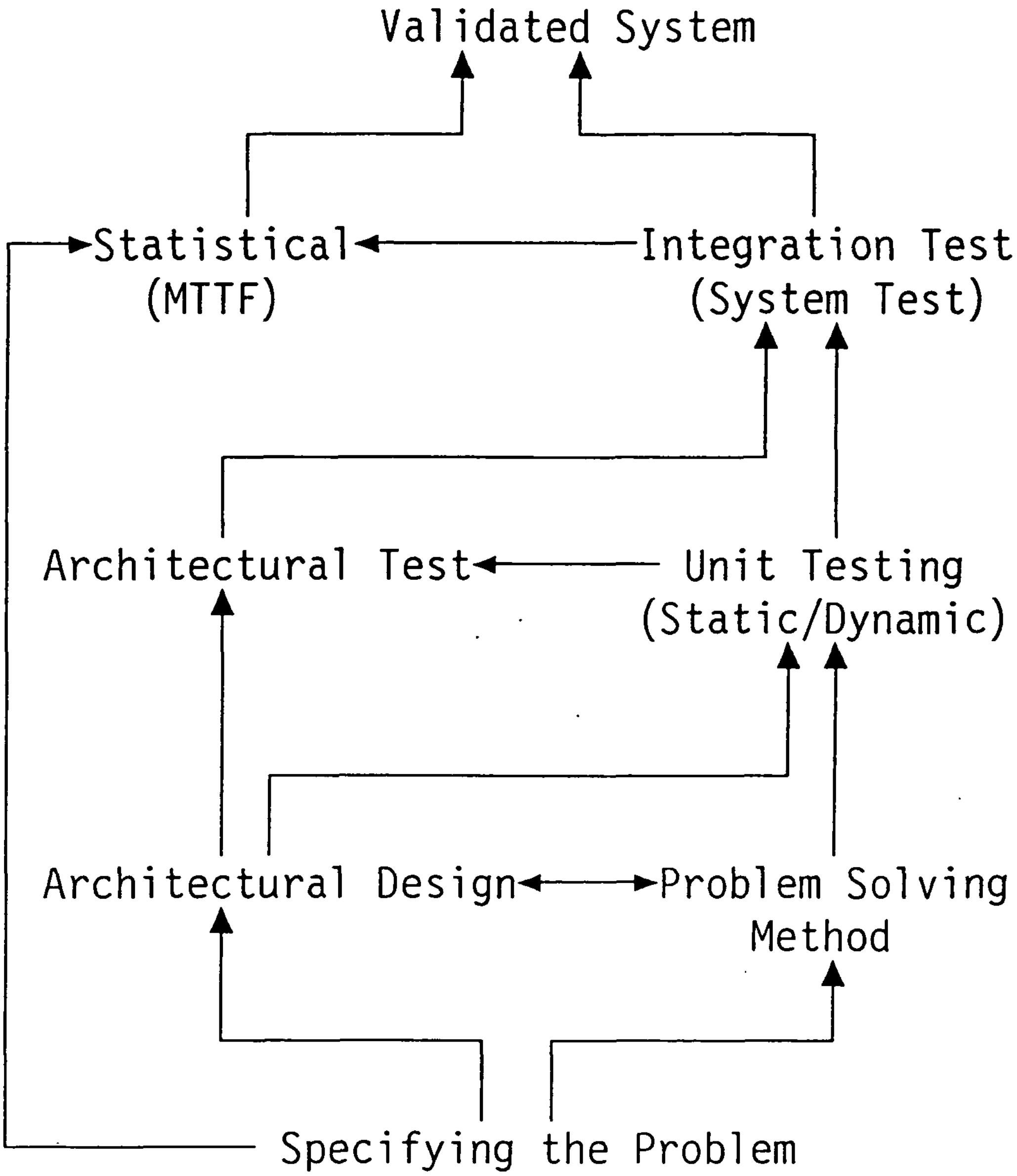
Unit A low-level function that when combined with other units in a controlled fashion implements the solution to a problem

Validation Showing that the completed software is a correct solution

Verification Showing that the software is being built correctly across all phases of a project

[1] These definitions reflect how these terms are used in the discussion that follows. They are not intended to reflect an industry accepted standard definition.

Validation Overview



Statistical

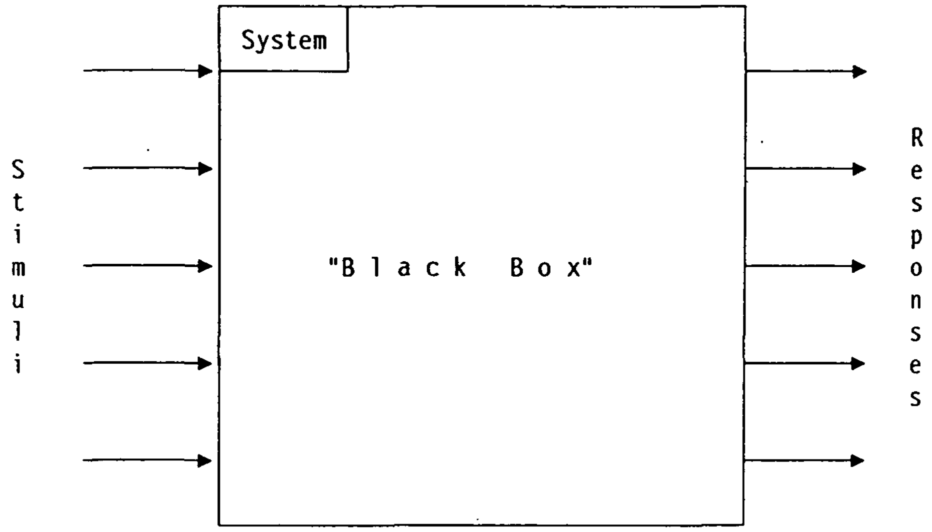
- Characteristics
  - Given the "system" is valid
    - Reliability should be predictable
    - Reliability should be measurable
    - Reliability should be repeatable
- Inputs
  - Collection of success criteria
    - What does it mean for the system to be acceptable?
  - Operational scenarios (nominal/off-nominal)
    - Validation test suite
  - Operating modes
  - Metrics defining the quality of the system
    - Mean-time-to-Failure (MTTF)

Statistical

- Implications
  - ► Easier to handle when a repository of test cases for a given product delivery exists
  - ► Easier when operational scenarios are identified (e.g., in requirements)
    - – driven by definition of succcess criteria

Integration/System Test

- Characteristics
  - ► Does the "system" do what it is supposed to?
    - – Is the response correct for each stimulus?
      - ↦ e.g., Running tests and evaluating responses
  - ► "How" a response is generated is not important at this level



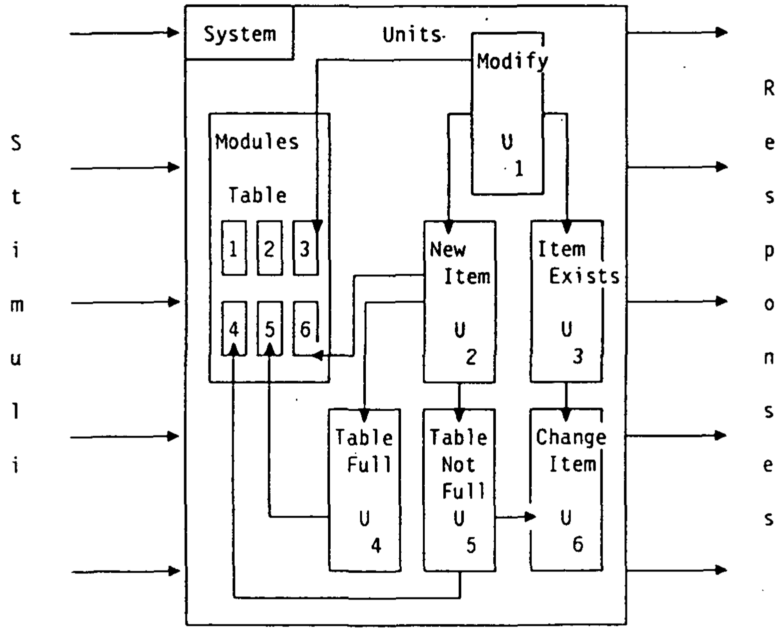
  - ► Stimuli tend to fall into "classes" or "groups"
  - ► Changes in stimulus result in response changes
    - – sensitivity to change
    - – demonstrate predictable behavior

Integration/System Test

- Inputs
  - ► Identification of state/function groups - units
    - — e.g. - in shuttle, there are many “principal” functions
  - ► Stimulus histories for each group
- Implications
  - ► Easier to analyze these “classes” separately with respect to the functions or data - i.e., unit testing
  - ► Only required to show that collecting the “pieces” to form the “system” has not introduced error

Unit Testing/Architectural Testing

- Characteristics
  - Given that "classes" of stimuli exist
    - the "system" can be viewed as containing units[2]
  - these units have stimuli and responses
    - therefore, they have subunits
  - the "system" has many "states"
  - some kinds of state are related



[2] This makes testing easier. This can be done regardless of how the system is actually implemented. For example, the Space Shuttle Flight Software (FSW) is tested by principal function even though this may not correspond to how it is implemented.

Unit Testing/Architectural Testing

- Characteristics
  - similarity within classes imply commonality
    - ↦ commonality can be partitioned out to reduce verification burden
    - ↦ lower level units/modules exist which are not explicit in the problem space
  - stimulus changes alter units required to generate a response
- Inputs
  - ► Specifications of expected behavior to achieve solution
    - For units
      - ↦ Stimulus histories for each unit
      - ↦ How a unit is related to another unit
      - ↦ Relationship to state (expected state transition)
    - For modules
      - ↦ Relationship of module to problem space
      - ↦ "Links" between different modules
      - ↦ Allowable state transitions

Unit Testing/Architectural Testing

- Implications
  - ► Design can be done in small steps by mapping conceptual entities to software entities - modularity
    - – can reduce the verification burden by easing the effects of changes to stimuli
    - – a design "architecture" can be defined to elicit module relationships
    - – bridges gap between problem and tested solution
      - ↦ modularity makes both jobs easier
  - ► A structured control mechanism exists for using "modules" to achieve desired function - problem solving method
  - ► A process of refining units into subunits exists
  - ► A process of "linking" modules together exists

Architectural Design

- Characteristics
    - differing levels of abstraction
    - information hiding
    - abstract interfaces
    - encapsulation
- Inputs
    - mapping between problem and solution spaces
- Implications
    - a thorough understanding of the problem is needed in order to define the mapping - "specifying the problem"
    - makes mapping from problem space to solution space explicit
    - makes the taxonomy of modules explicit

Problem Solving Method

- Characteristics
  - ► units control other units to accomplish their function
    - – is the method appropriate for the problem?
    - – is the method acceptably efficient?
  - ► a mapping exists between units of function and "what" the system is supposed to do
  - ► units interact with "state"
- Inputs
  - ► Allowable state transitions
  - ► Stimulus histories
  - ► Mapping to the problem being solved
- Implications
  - ► a thorough understanding of the problem is needed in order to define the mapping and refinement
  - ► structured refinement reduces complexity of control structure
  - ► limiting the kind of structures used eases analysis
  - ► how to interface with elements of the problem space that have been captured as modules

Specifying the Problem

- Characteristics
  - complete description of the problem to be solved
    - direct correlation to stimuli (operating environment)
    - direct correlation to response (what to do in that environment)
  - complex systems are too hard to understand taken all at once
- Inputs
  - customer wants/desires
  - knowledge of experts in the application domain
- Implications
  - partitions are necessary to break-down complexity
    - refinement
    - abstraction
  - documenting all this is a major key to success
    - should describe the problem space while leaving freedom for the designer to pick the appropriate solution
    - should be traceable

The Traffic Controller Problem

- Consider the following problem:

  A simple traffic light controller at a four way intersection has car arrival sensors and pedestrian crossing buttons. In the absence of car arrival and pedestrian crossing signals, the traffic light controller switches the direction of traffic flow every 2 minutes. With a car or pedestrian signal to change the direction of traffic flow, the reaction depends on the status of the auto and pedestrian signals in the direction of traffic flow; if auto pedestrian sensors detect no approaching traffic in the current direction of traffic flow, the traffic flow will be switched in 15 seconds, if such approaching traffic is detected, the switch in traffic flow will be delayed 15 seconds with each new detection of continuing traffic up to a maximum of one minute.

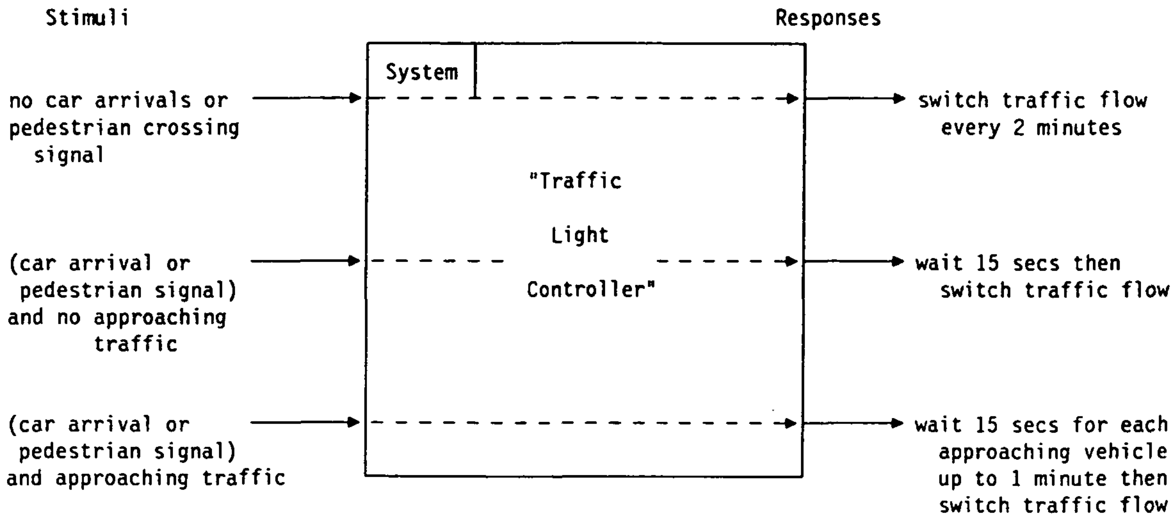
- Take a few minutes and write down the key tasks the "traffic controller" is to do

- Exchange your descriptions with a neighbor and then spend a few minutes deciding how well their description fits your understanding of the "traffic controller"

  - is this a testable description of the system?

The Traffic Controller Problem

- Initial “black box” view of system testing



Stimuli

Responses

System

"Traffic Light Controller"

no car arrivals or pedestrian crossing signal → switch traffic flow every 2 minutes

(car arrival or pedestrian signal) and no approaching traffic → wait 15 secs then switch traffic flow

(car arrival or pedestrian signal) and approaching traffic → wait 15 secs for each approaching vehicle up to 1 minute then switch traffic flow

The Traffic Controller Problem

- Refine Requirements based on further understanding of the problem
- State becomes evident
  - See ***Refinement of the initial 'black-box' view***
  - What is the color of the light in a given direction?
  - How long has the controller waited to switch the light?
- State helps identify and classify stimulus/response histories
  - See ***Identification of 'classes' based on state***
  - The state remaining the same might imply testing one scenario verifies the other scenario as well
- Continuing this refinement will lead to a more organized test approach
  - operational scenarios can be constructed/selected

The Traffic Controller Problem

- Refinement of the initial ‘black-box’ view

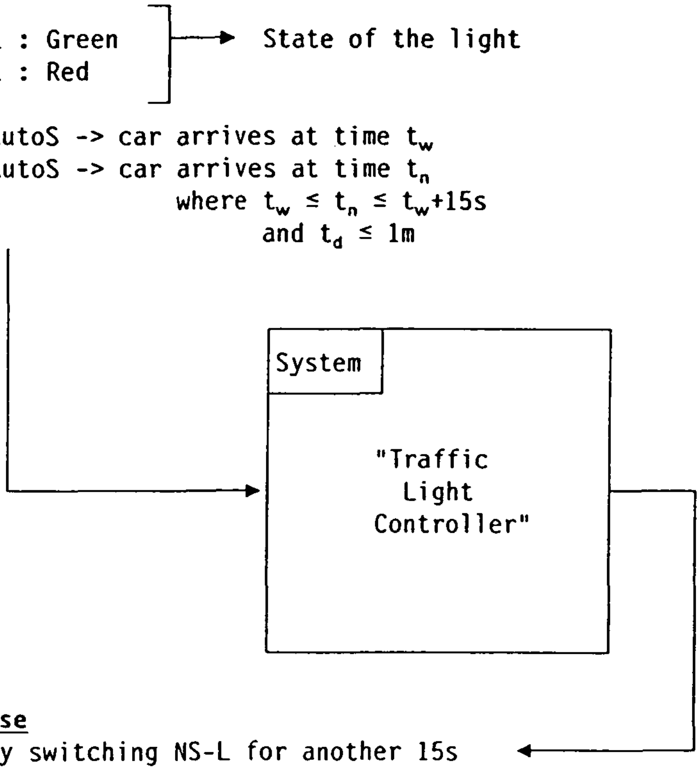
Notation:

WE : West-East
NS : North-South
L: Light
AutoS: Auto sensor
PedS : Pedestrian sensor
t_d : time controller has delayed since sensor

Stimulus

NS-L : Green
WE-L : Red
→ State of the light

WE-AutoS -> car arrives at time t_w
NS-AutoS -> car arrives at time t_n
where $t_w \le t_n \le t_w$+15s
and $t_d \le$ 1m



System

"Traffic Light Controller"

Response
delay switching NS-L for another 15s

The Traffic Controller Problem

- Identification of 'classes' based on state

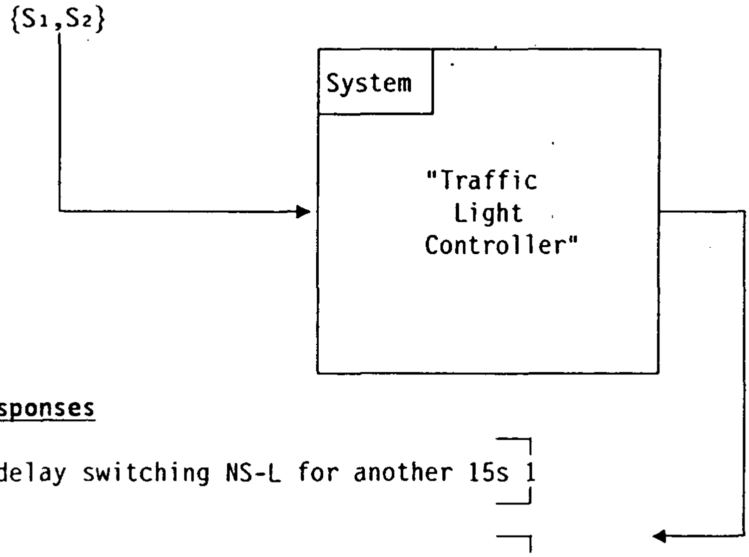
Stimuli

NS-L : Green, WE-L : Red → State of the light

1 WE-AutoS -> car arrives at time t_w
NS-AutoS -> car arrives at time t_n
where $t_w \le t_n \le t_w + 15s$
and $t_d \le 1m$

NS-L : Green, WE-L : Red → State of the light

2 WE-PedS -> pedestrian arrives at time t_w
NS-AutoS -> car arrives at time t_n
where $t_w \le t_n \le t_w + 15s$
and $t_d \le 1m$

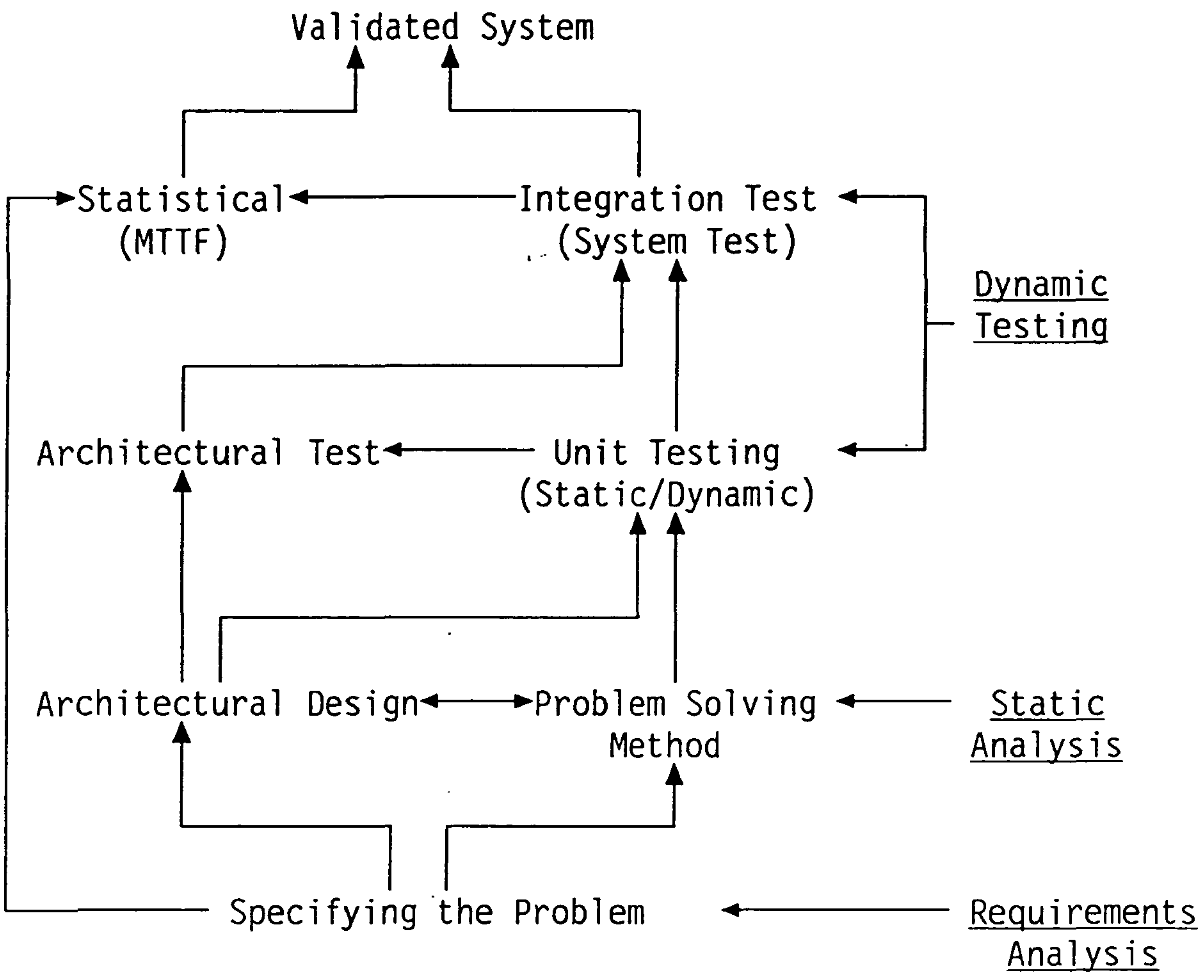


Responses

delay switching NS-L for another 15s 1

delay switching NS-L for another 15s 2 {R1,R2}

Validation Techniques Overview



Integration/System Testing

- Criteria for Selecting Methods
    - ► Techniques that do not dependent on viewing the system's internal structure are required
        - — Internal structure is hidden due to the "black-box" view
    - ► Active Interface Testing
- Methods
    - ► Realistic Testing[2]
    - ► Stress Testing[2]
    - ► Functional Testing[2]
        - — Box-structured analysis
        - — Specification-based testing[1]
    - ► Performance Testing[2]
    - ► Metric-based Testing[2]
    - ► Statistical Record-keeping[2]
    - ► Active Interface Testing[2]

Statistical

- Criteria for Selecting Methods
  - Same criteria as Integration/System Testing, plus ...
  - Techniques that focus on identifying operational scenarios that provide statistically random sampling
- Methods
  - Random Testing[2]
  - Regression Testing[2]
  - Software Reliability Estimation[2]
  - Realistic Testing[2]

Unit Testing/Architectural Testing

- Criteria for Selecting Methods
  - Techniques must provide for analysis of the structure of the system - both from a data and function perspective
    - Internal structure is visible due to "white-box" view
  - Techniques must provide for analysis of integrating units and modules
- Methods
  - Reliability Testing[2]
  - Structural Testing[2]
  - Mutation Testing[2]
  - Error-introductionTesting [2]
  - Functional Testing[2]

Architectural Design

- Criteria for Selecting Methods
  - ► Techniques should show that the design maps to the problem
  - ► Techniques must be "static" in nature
    - – nothing "executable" has been built at this point
- Methods
  - ► Data Analysis'
  - ► Inspections (Formal or Walkthrough)'
  - ► Input Space Partitioning'
    - – Domain analysis'
    - – Partition analysis'
  - ► Object-Oriented Analysis
  - ► Entity/Relation Analysis

Problem Solving Method

- Criteria for Selecting Methods
  - Techniques should elicit levels of refinement
  - Techniques should apply to showing that a specification is correct
- Methods
  - Inspections (Formal or Walkthrough)[1]
  - Defect Analysis[2]
  - Control Analysis[2]
    - Structured Analysis[1]
    - Data-Flow Analysis[1]
    - Stepwise Refinement
  - Algorithm Analysis[1]
    - Symbolic Execution
    - Mathematical Theorem proving
  - Design Simulation[2]
  - Design Compliance Analysis[2]

Specifying the Problem

- Criteria for Selecting Methods
  - Methods should be selected that ability to comprehend the problem to be solved
  - Methods should be chosen that provide a format that is easily traceable in later tasks
- Methods
  - Requirements Language Analysis[2]
  - Requirements Language Processing[2]
  - Mathematical Verification of Requirements[2]
  - Formal Requirements Review[2]
  - Requirements Tracing/Traceability Analysis[2]

References

1. Boeing Aerospace Company. “Software Test Guide.” Software Test Handbook Rome Air Development Center Report RADC-TR-84-53. Air Force Systems Command, Griffis Air Force Base, NY: March 1984.

2. Science Applications International Corporation. “Task 1: Review of Conventional Methods.” Guidelines for Verification and Validation of Expert Systems. Document # SAIC-91/6660, April 19, 1991.